

Parallel and Distributed Computing

Chapter II

Models of Parallel Computing

Prof. Dr. Ronald Moore



Last Modified: 4/15/12

Outline of Chapter II – Models

- I Introduction
- II Models of Parallel Computing
- III Parallel Computation Design
- IV The Message-Passing Paradigm
- V The Shared Memory Paradigm
- VI Frontiers
- VII Summary

- A) Jargon & Terminology
- B) Taxonomies:
 - 1) Flynn's
 - 2) R. C. Moore's
- C) Models:
 - 1) Theoretical
 - 2) Hardware
 - a) Command Flow
 - b) Memory Models
 - 3) Communication
- D) Summary

Jargon & Terminology (1)

Parallelism: The subject of this course...

- **Quasi-Parallelism:** Time-sharing. Not really parallel.
- **Parallelism:** More than one operation can be performed at the same time.
- **Concurrency:** Either Quasi-Parallelism or “real” Parallelism.
- **Sequential:** The opposite of concurrent.

Jargon & Terminology (2)

Ontology (the *things* that survive Occam's Razor)

- **Program:** A self-contained series of instructions (and data definitions).
- **Process:** The execution of a program; a program being run (including it's data, state and resources)
- **Thread:** A light-weight process – a sequence of operations being executed. A process can have more than one thread, but a thread cannot have more than one process.

Occam's Razor:
*Entia non sunt
multiplicanda praeter
necessitatem,*
Entities must not be
multiplied beyond
necessity
(William of Ockham,
14th century)

Jargon & Terminology (3)

Ontology (continued)

- **Task:** A program or a portion of a program. A sequence of instructions that operate together. A program can be broken into tasks (but a task cannot be broken into programs). A term used in algorithm design & development.
- **Unit of Execution (UE):** The execution of a task – either a process, or a thread, or a portion of a process or a thread.
- **Processing Element (PE):** The hardware element which executes a UE. The size and capabilities of a PE can vary widely – from an entire workstation to one core (on a multicore CPU) to a component inside one core.

Jargon & Terminology (4)

The Limits of Parallelism

- **Dependencies:** A relationship between two operations (tasks) when prevents (or limits) their running in parallel:
 - ➔ **Data Dependency:** When one operation requires (as input) data produced by another.
 - ➔ **Procedural (Branch) Dependency:** When the results of one operation determine whether (or what) another operation will execute.
 - ➔ **Resource Dependency:** When one operation requires a resource (e.g. a UE or part of a UE) data currently being used by another. One form of “false” dependency.

Jargon & Terminology (5)

The Limits of Parallelism (continued)

- **Synchronization:** Anything that creates a (reliable, predictable, deterministic) relationship between the operation of two parallel tasks.
 - **Asynchronous:** Two tasks that have little or no synchronization are called asynchronous.
 - **Synchronous:** Two tasks that are closely coupled are called asynchronous. Note however that these are not absolute categories, but rather two ends of a spectrum.
 - **Race Condition:** What happens when synchronization goes wrong (usually, when synchronization is missing). An error which happens sometimes, depending on the timing of events thought (erroneously) to be independent.
 - **Deadlock, Livelock:** Two extreme forms of Race Conditions, where two or more tasks cannot (effectively) proceed.

Outline of Chapter II – Models

- I Introduction
- II Models of Parallel Computing
- III Parallel Computation Design
- IV The Message-Passing Paradigm
- V The Shared Memory Paradigm
- VI Frontiers
- VII Summary

- A) Jargon & Terminology
- B) Taxonomies:
 - 1) Flynn's
 - 2) R. C. Moore's
- C) Models:
 - 1) Theoretical
 - 2) Hardware
 - a) Command Flow
 - b) Memory Models
 - 3) Communication
- D) Summary

Flynn's Taxonomy (1)

Why taxonomies? Taxonomies have helped other fields confronted by overwhelming diversity, so maybe they can help us!

Flynn's Taxonomy:

- Is the most famous taxonomy over parallel computation.
- Proposed by Michael J. Flynn.
- Assumes that every (interesting) computer operates on a **stream of instructions** (whose bandwidth is either *single* or *multiple*)...
- ...and a **stream of data** (whose bandwidth is either *single* or *multiple*).
- Thus there are 4 possible combinations...

Flynn's Taxonomy (2)

SISD: Single Instruction, Single Data Stream:

A sequential Computer.

Pronounced
"sim-dee"

SIMD: Single Instruction, Multiple Data Stream:

A *Vector* Computer.

Often pronounced dead, often rediscovered (used in DSPs, GPGPUs...). Example Usage: Multiple every element in an array by 2.

MISD: Multiple Instruction, Single Data Stream:

Only theoretically possible, right?
Or perhaps for pipelines.
Or redundant (heterogenous) systems (e.g. NASA).
Or Very Long Instruction Word (VLIW – see below)?

MIMD: Multiple Instruction, Multiple Data Stream:

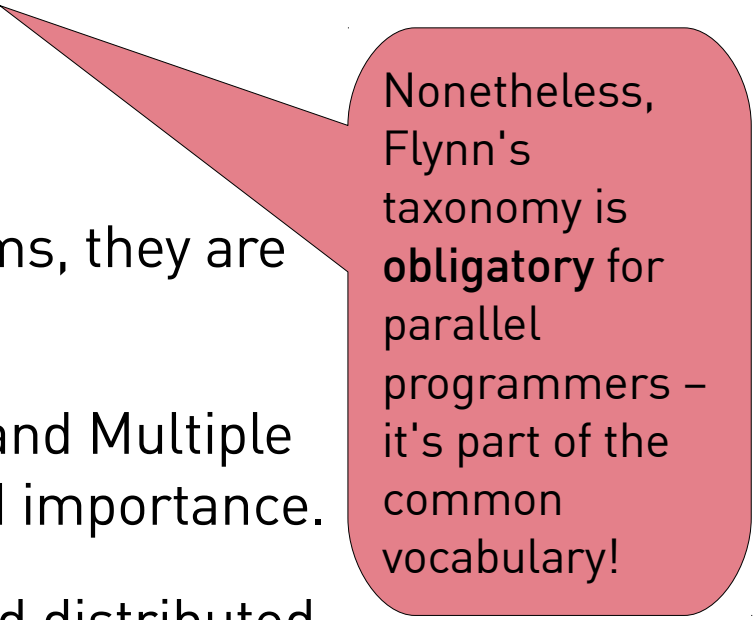
A parallel computer where every PE can be separately programmed. See Chapter 1 for examples.

Pronounced
"mim-dee"

Flynn's Taxonomy (3)

Problems with Flynn's Taxonomy:

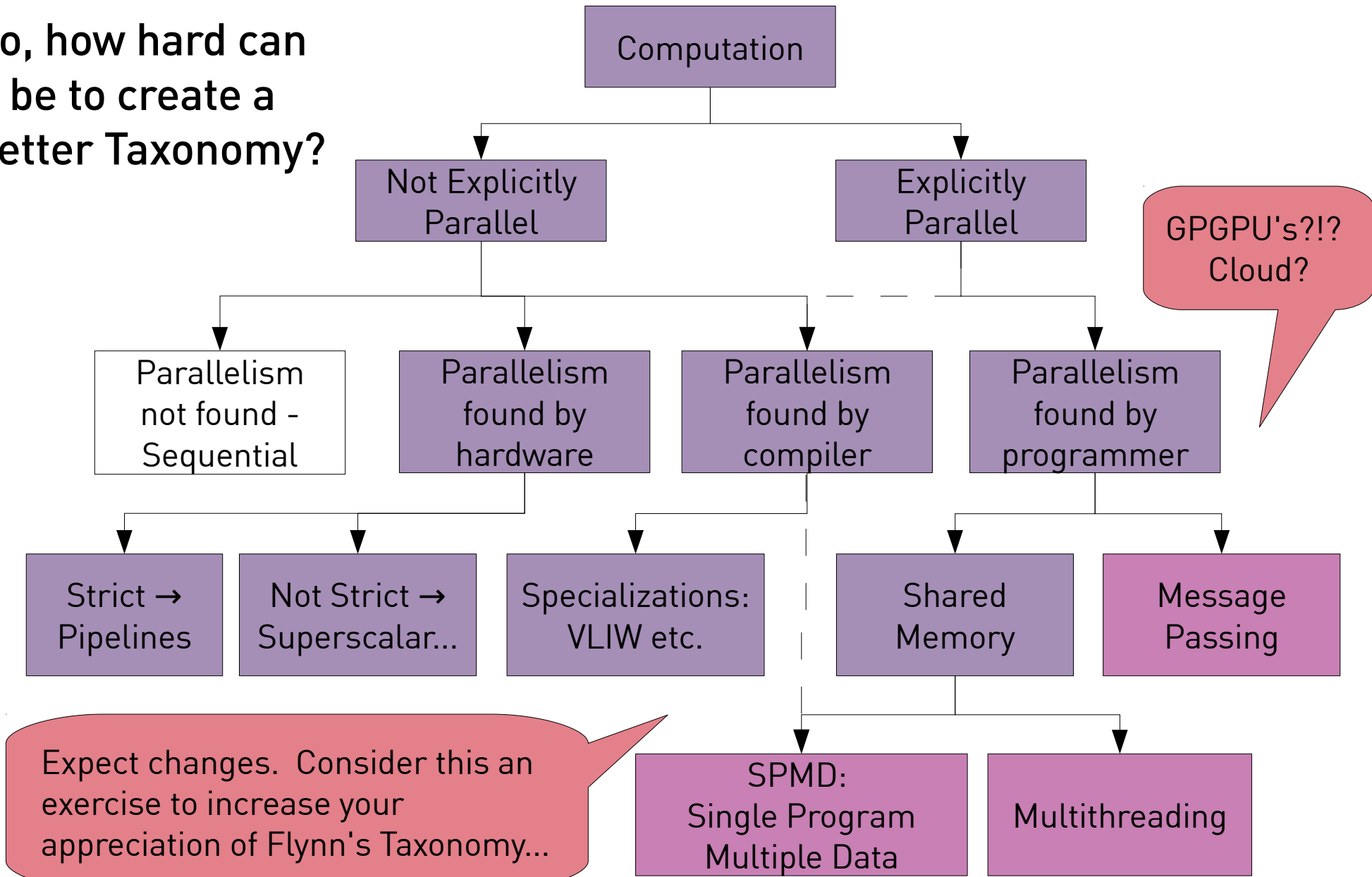
- The MISD category remains a mystery.
- Data and Instructions do not come in streams, they are read from *memory* in *cache lines*.
- The distinction between Single Instruction and Multiple Instruction Stream receives an exaggerated importance.
- The distinction between shared memory and distributed memory is not expressed.
- It leaves no room for compromises such as SPMD – *Single Program, Multiple Data* (synchronicity is a spectrum!).
- The taxonomy is hardware-centric and not software-centric or particularly useful for designing algorithms.



Nonetheless, Flynn's taxonomy is **obligatory** for parallel programmers – it's part of the common vocabulary!

R. C. Moore's Taxonomy (1)

So, how hard can
it be to create a
better Taxonomy?



R. C. Moore's Taxonomy (2)

What are the distinctions that matter?

- What kind of programming language?
 - ➔ Totally conventional, e.g. C, Fortran...?
 - ➔ Extended conventional, e.g. MPI, OpenMP?
 - ➔ Explicitly parallel – see C. A. R. Hoare's CSP?
 - ➔ Agnostic: Functional languages like Haskell?
- How do the UE's cooperate?
 - ➔ By sharing data structures – i.e. Shared Memory?
 - ➔ By exchanging Messages – Message Passing?
 - ➔ Note that Hardware is not Software –
 - ★ you can build a *distributed shared memory*,
 - ★ or pass messages in a shared (hardware) memory

R. C. Moore's Taxonomy (3)

What are the distinctions that matter? (concluded)

- **Granularity:**

- ➔ Most hardware is transparently parallel (out-of-order execution, deep pipelines, etc.) → microscopic granularity.
- ➔ Some hardware allows the compiler and/or programmer to direct the parallelism → fine grained.
- ➔ Shared memory is faster (usually) than networks, so shared memory allows finer granularity than message passing → coarse grained.
- ➔ Replicated hardware is most easily programmed with separate processes on each PE → coarser grained.
- ➔ This is a spectrum, not a categorization. Finding the right granularity is a function of both platform and application.

Outline of Chapter II – Models

- I Introduction
- II Models of Parallel Computing
- III Parallel Computation Design
- IV The Message-Passing Paradigm
- V The Shared Memory Paradigm
- VI Frontiers
- VII Summary

- A) Jargon & Terminology
- B) Taxonomies:
 - 1) Flynn's
 - 2) R. C. Moore's
- C) Models:
 - 1) Theoretical
 - 2) Hardware
 - a) Command Flow
 - b) Memory Models
 - 3) Communication
- D) Summary

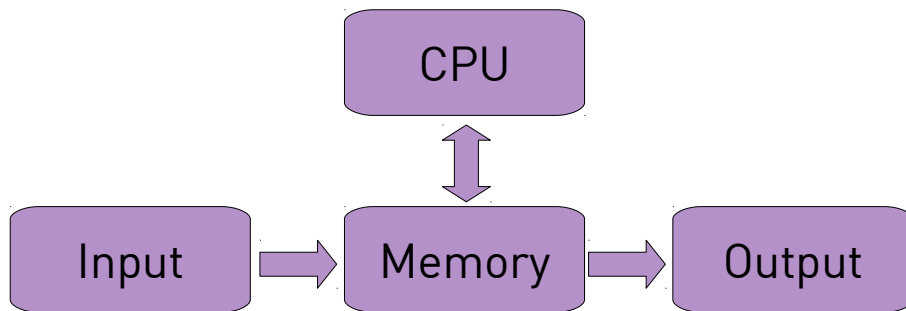
The Theoretical Model – PRAM (o)

Well, if taxonomies don't capture parallelism, how about a model?

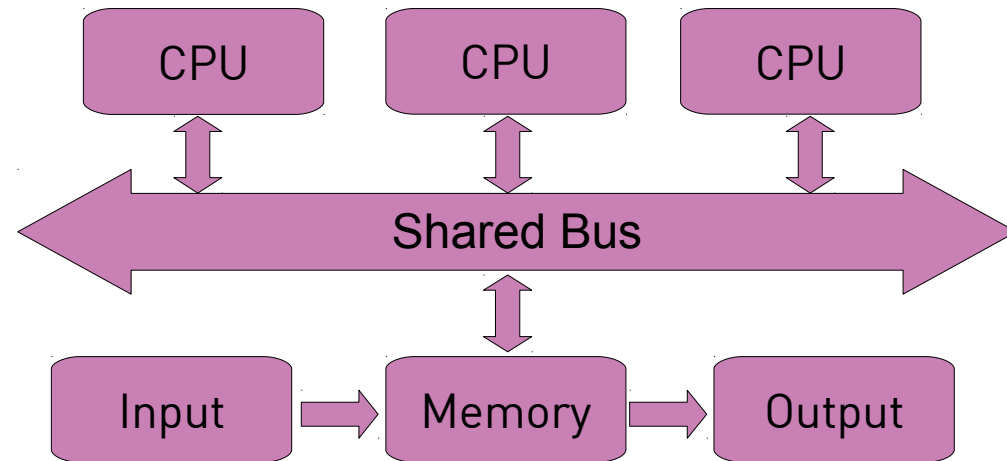
- **PRAM = Parallel Random Access Machine:**

- ➔ Extends the theoretical Random Access Machine (RAM)

Essential:
Infinitely many
CPUs!



RAM = Random Access Machine



PRAM = Parallel Random Access Machine

The Theoretical Model – PRAM (1)

- **PRAM = Parallel Random Access Machine:**

- ➔ Extends the theoretical Random Access Machine (RAM)
- ➔ Used for complexity theory – deriving “ $O(n)$ ” complexity classes.
- ➔ *Not* included: communication costs, synchronization physical limits on the number of PEs, size of memory, speed of network...

- **4 Types of PRAM**

- | | |
|--|---|
| ➔ EREW – Exclusive Read,
Exclusive Write | ➔ ERCW – Exclusive Read,
Concurrent Write |
| ➔ CREW – Concurrent Read,
Exclusive Write | ➔ CRCW – Concurrent Read,
Concurrent Write |

The Theoretical Model – PRAM (2)

So, how does *concurrent write* work?

- 4 Types of Concurrent Write

- ➔ *Common* – All write the same value (anything else is illegal!)

- ➔ *Arbitrary* – “Winner” is determined non-deterministically

- ➔ *Priority* – A predictable scheme determines “winner” (i.e. leftmost PE wins).

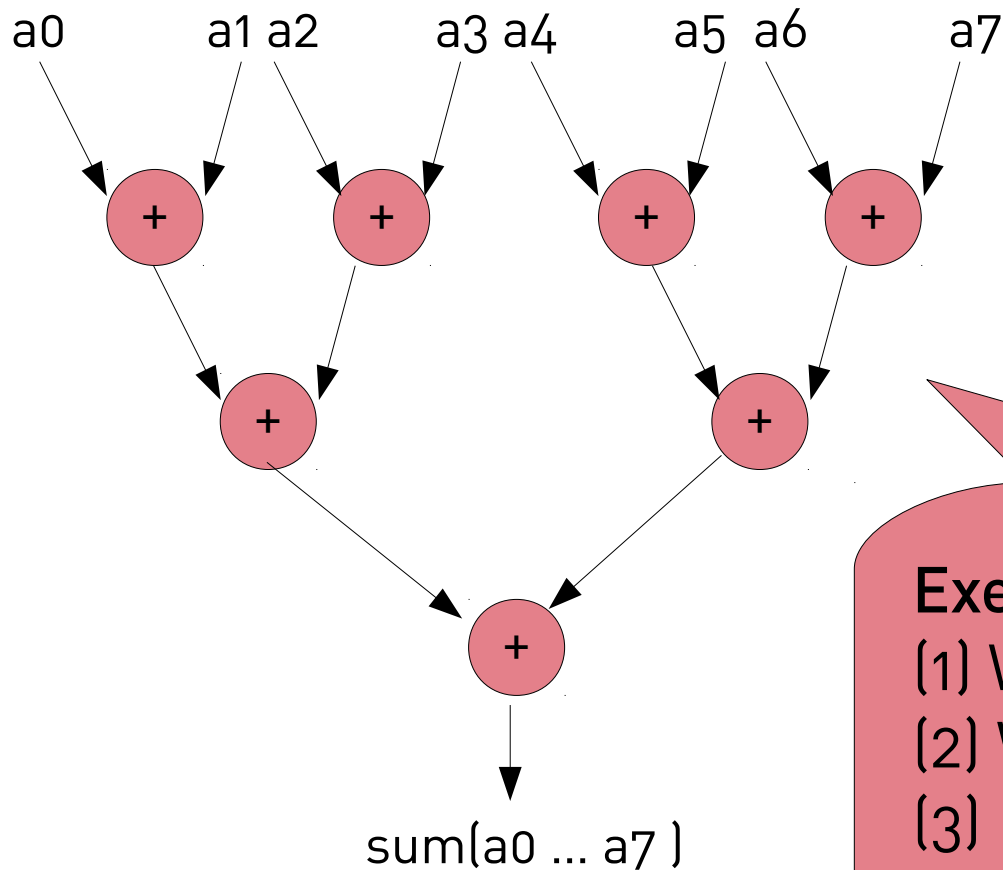
- ➔ *Other* – The result is the sum of the values written, or the logical AND, or the logical OR, or...

Exercise:

- (1) What is the complexity of “naïve” matrix multiplication on a PRAM?
- (2) Does it matter which kind of PRAM we use?

Another Theoretical Model – Dataflow Graphs (1)

Dataflow graphs visualize dependencies, parallelism.

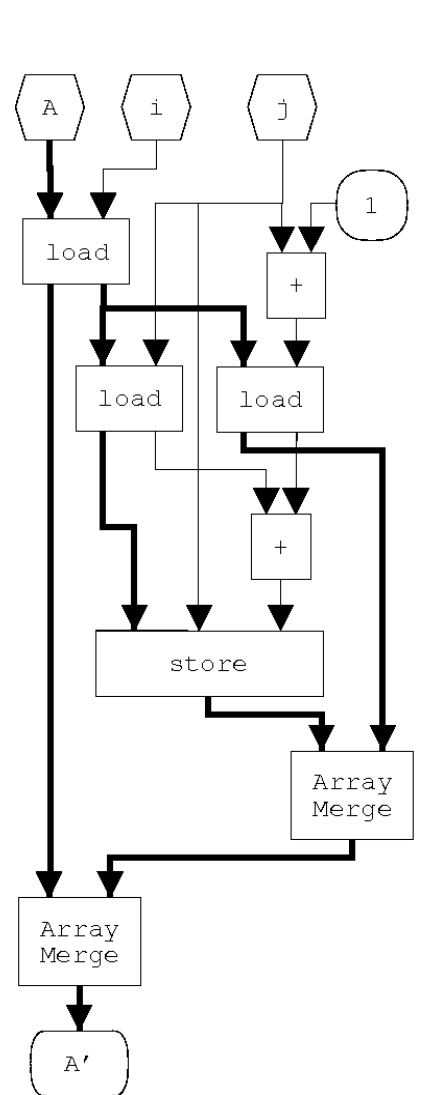


Exercises:

- (1) Where is the parallelism?
- (2) What determines run time?
- (3) How can we express a loop?
Or other control structures?

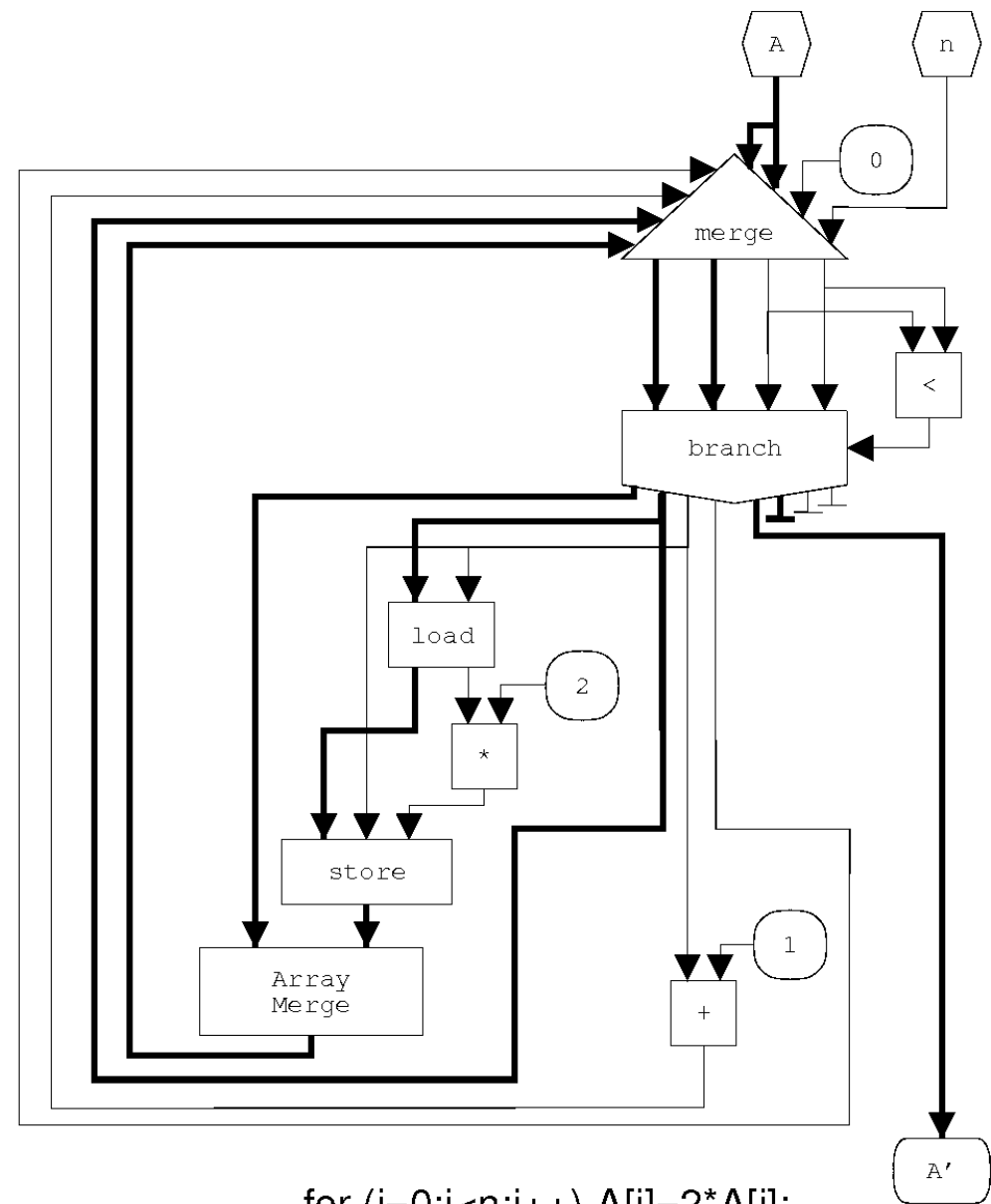
Another Theoretical Model – Dataflow Graphs (2)

More complicated DFG's...



$$A[i][j] = A[i][j] + A[i][j+1]$$

(a)



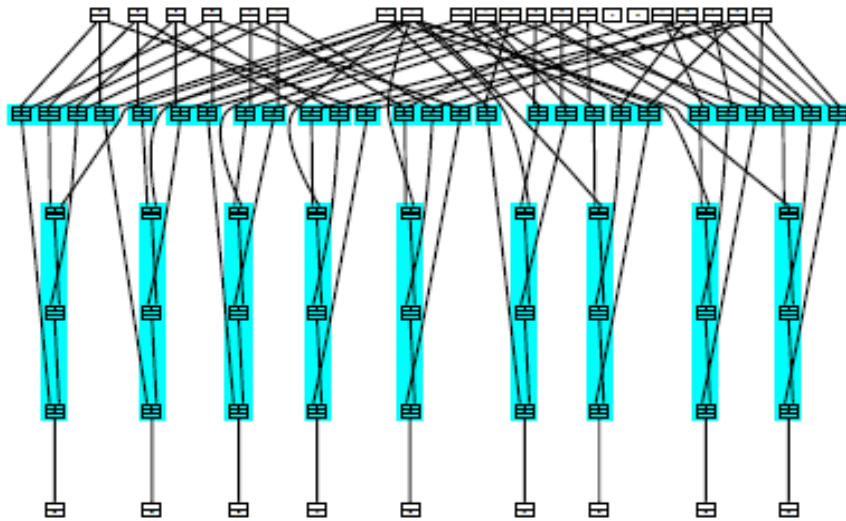
for (i=0; i<n; i++) A[i] = 2*A[i];

(b)

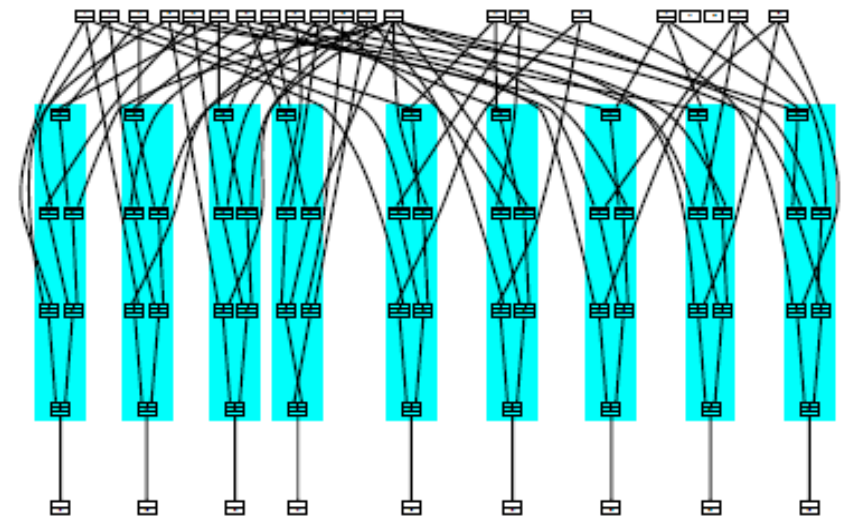
Sources: Ronald Moore,
*SDAARC: A Self
Distributing Associative
Architecture*, Shaker
Verlag, 2002

Another Theoretical Model – Dataflow Graphs (3)

Dataflow can be used to find parallelism, extract UEs, and map these to PEs. Example: 3x3 Matrix Multiplication.



Extremely low communication costs
 $\Rightarrow 36 \mu\text{Threads}$



Higher communication costs
 $\Rightarrow 9 \mu\text{Threads}$

Sources: Ronald Moore, *SDAARC: A Self Distributing Associative Architecture*, Shaker Verlag, 2002 and/or Ronald Moore, Melanie Klang, Bernd Klauer, and Klaus Waldschmidt, "Hybrides Scheduling". In *ARCS '99: Architektur von Rechensystemen 1999; Vorträge der Workshops im Rahmen der 15. GI/ITG Fachtagung*, pages 211–218, Jena, Oct. 1999. Universität Jena, Institut für Informatik.

Outline of Chapter II – Models

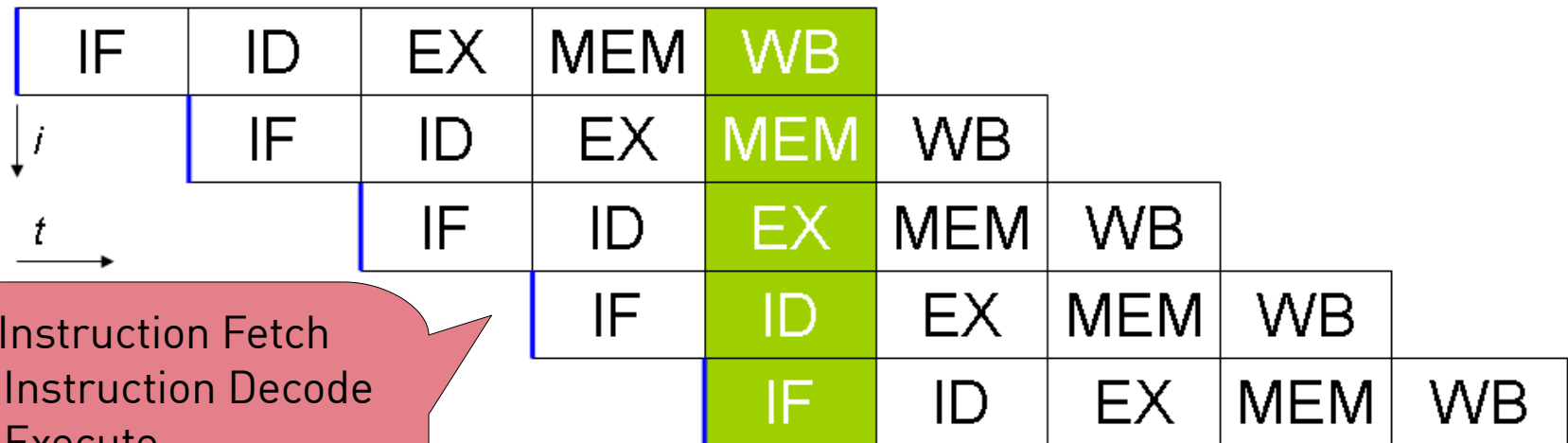
- I Introduction
- II Models of Parallel Computing
- III Parallel Computation Design
- IV The Message-Passing Paradigm
- V The Shared Memory Paradigm
- VI Frontiers
- VII Summary

- A) Jargon & Terminology
- B) Taxonomies:
 - 1) Flynn's
 - 2) R. C. Moore's
- C) Models:
 - 1) Theoretical
 - 2) Hardware
 - a) Command Flow
 - b) Memory Models
 - 3) Communication
- D) Summary

Hardware Models: Pipelining

What can hardware teach us about parallelism?

Pipelining (Example: Classic 5-stage RISC Pipeline)



- 1) Instruction Fetch
- 2) Instruction Decode
- 3) Execute
- 4) Memory Access
- 5) Writeback

Speedup determined by

- We can increase *throughput* even when data dependencies keep us from exploiting parallelism for any given output.

Remedies?
Branch-Prediction,
Loop-Unrolling...

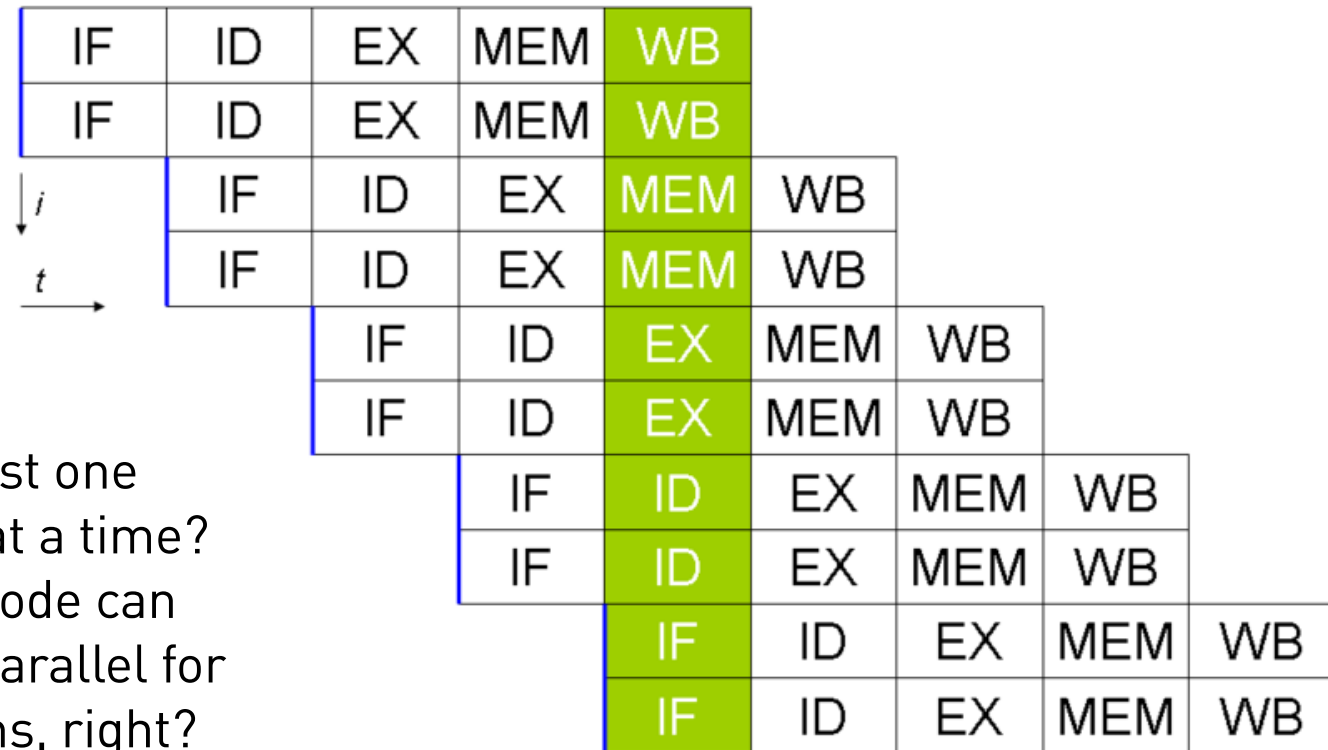
- ★ Number of stages
- ★ Duration of slowest stage
- ★ *Hazards* (e.g. cache misses, branches...)

Source: "Classic RISC pipeline." *Wikipedia, The Free Encyclopedia*. <http://en.wikipedia.org/w/index.php?title=Classic_RISC_pipeline&oldid=306463830>.

Hardware Models: Superscalar (1)

What can hardware teach us about parallelism?

Superscalar = Multiple Pipelines (Here 2xSuperscalar)



→ Why fetch just one Instruction at a time? Fetch & Decode can proceed in parallel for n instructions, right?

Really?
(When) Will this work?

Source: "Superscalar." *Wikipedia, The Free Encyclopedia*. 27 Aug 2009, <<http://en.wikipedia.org/w/index.php?title=Superscalar&oldid=310378205>>.

Hardware Models: Superscalar (2)

Consider these three *equivalent* code segments:

1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000

(i)

1. load R1, @1000
2. add R1, @1004
3. add R1, @1008
4. add R1, @100C
5. store R1, @2000

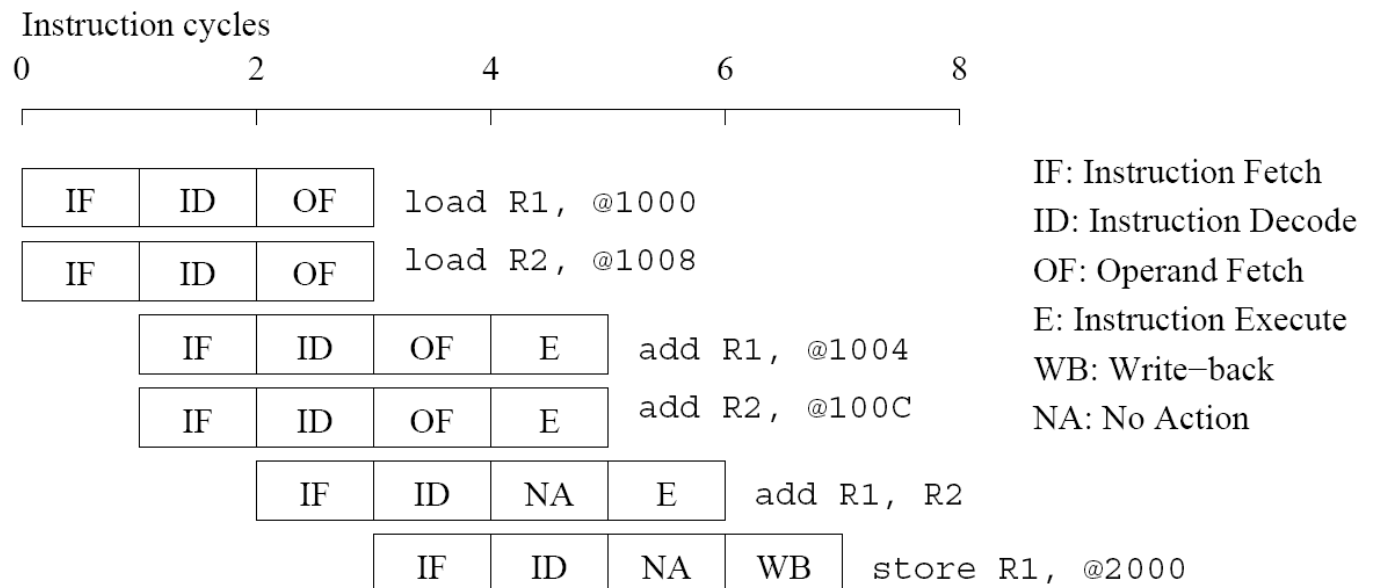
(ii)

1. load R1, @1000
2. add R1, @1004
3. load R2, @1008
4. add R2, @100C
5. add R1, R2
6. store R1, @2000

(iii)

How will the other two fragments run? What can the architecture do to improve on this?

(a) Three different code fragments for adding a list of four numbers.



Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

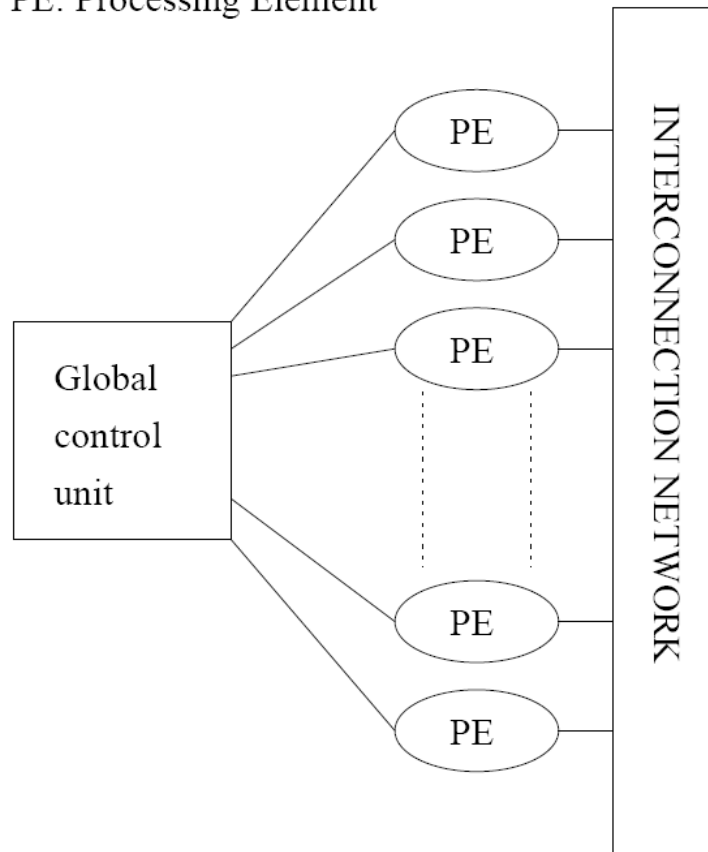
(b) Execution schedule for code fragment (i) above.

Hardware Models: Vector (SIMD 1)

Why should dependencies be found by the hardware and not the compiler?

Possibility 1: SIMD

PE: Processing Element



(a)

Figure 2.3 A typical SIMD architecture (a)

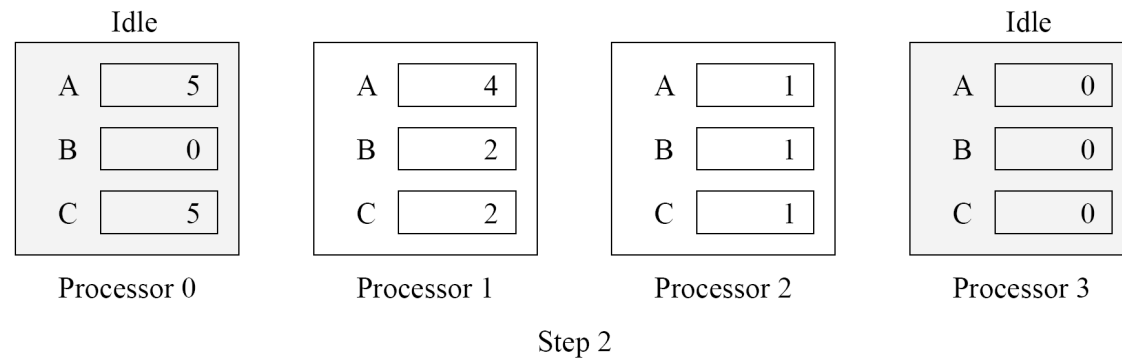
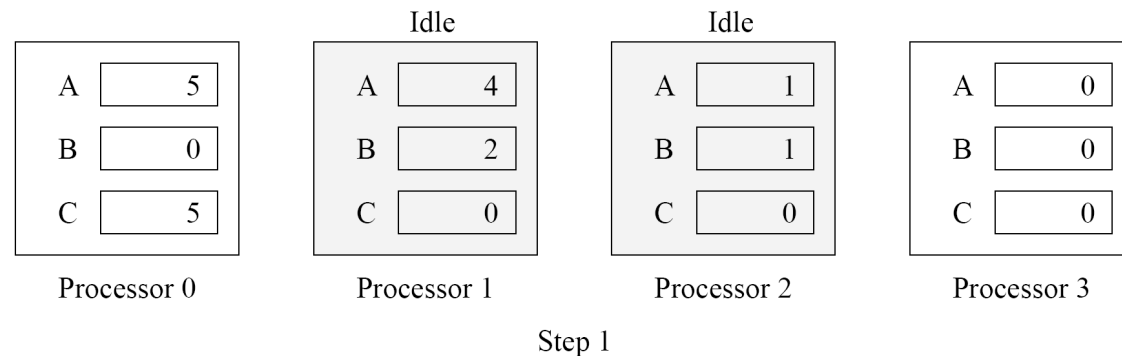
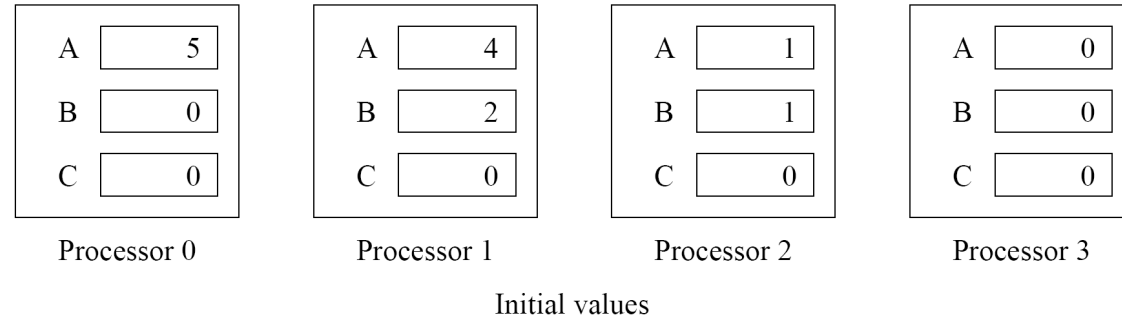
Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Hardware Models: Vector (SIMD 2)

```

if (B == 0)
    C = A;
else
    C = A/B;
    
```

(a)



(b)

Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Figure 2.4 Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

Hardware Models: Vector (VLIW)

Why should dependencies be found by the hardware and not the compiler?

Possibility 2: Very Long Instruction Word (VLIW)

Instruction

$n-1$
n	BRANCH	ADD	ADD	MUL
$n+1$

Is this SIMD or MIMD or MISD for that matter?

Compare Intel & HP's EPIC, the Itanium

“...the “Itanium” approach that was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write.”

Donald E. Knuth and Andrew Binstock, *Interview with Donald Knuth*, April 25, 2008,
<http://www.informit.com/articles/article.aspx?p=1193856>

Hardware Models: Hyperthreading

Why should dependencies be found in one process and not *between* processes?

**Possibility 1:
Simultaneous
Multithreading (SMT)**

Intel calls this
“Hyperthreading”

**Possibility 2:
Multicore CPUs**

**Possibility 3:
Multicore CPUs with
SMT**

Simultaneous Multithreading:

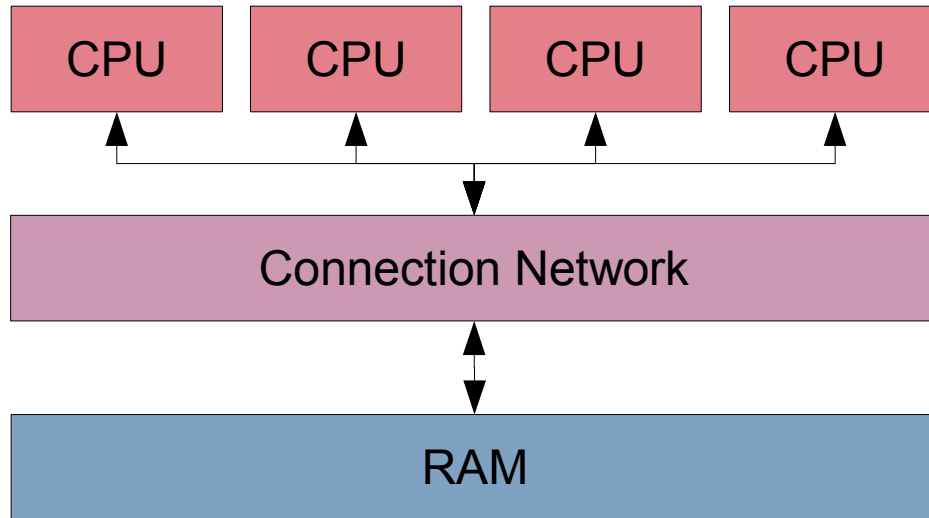
- ➔ **Hazard:** When one process (thread) has dependencies which limit the number of instructions that can be issued simultaneously (data, resource, control hazards or cache misses).
- ➔ **Solution:** Issue instructions from *other* processes (threads).
- ➔ **Problem:** Each thread wants to use the same registers.
- ➔ **Solution:** Register renaming. My register R1 is not the same register as your register R1...
- ➔ **Appearance:** One CPU “looks like” more than one CPU to the operating system.

Outline of Chapter II – Models

- I Introduction
- II Models of Parallel Computing
- III Parallel Computation Design
- IV The Message-Passing Paradigm
- V The Shared Memory Paradigm
- VI Frontiers
- VII Summary

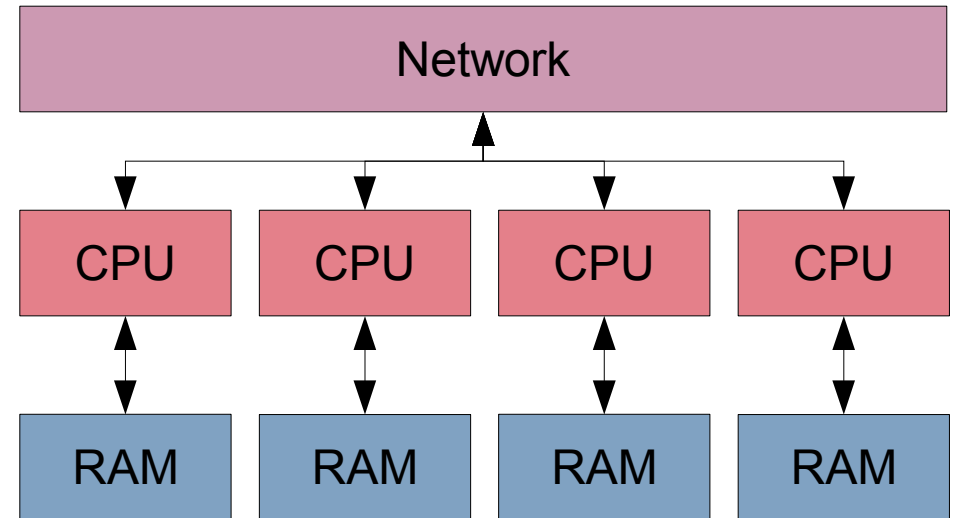
- A) Jargon & Terminology
- B) Taxonomies:
 - 1) Flynn's
 - 2) R. C. Moore's
- C) Models:
 - 1) Theoretical
 - 2) Hardware
 - a) Command Flow
 - b) Memory Models
 - 3) Communication
- D) Summary

Memory Models: UMA vs. NUMA



UMA = Uniform Memory Access
= physically shared memory.

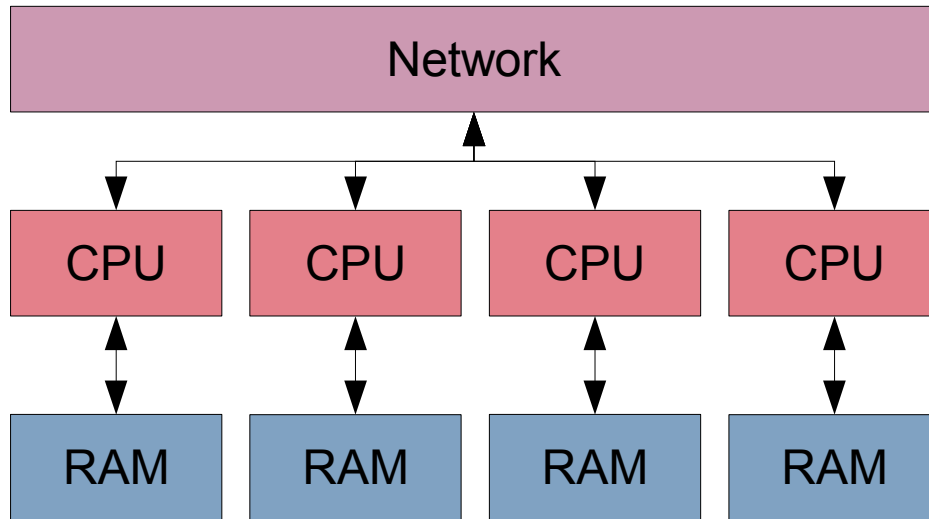
Also known as SMP
= Symmetric
Multiprocessor
= Shared Memory
Processor



NUMA = Non-Uniform Memory
Access = distributed memory

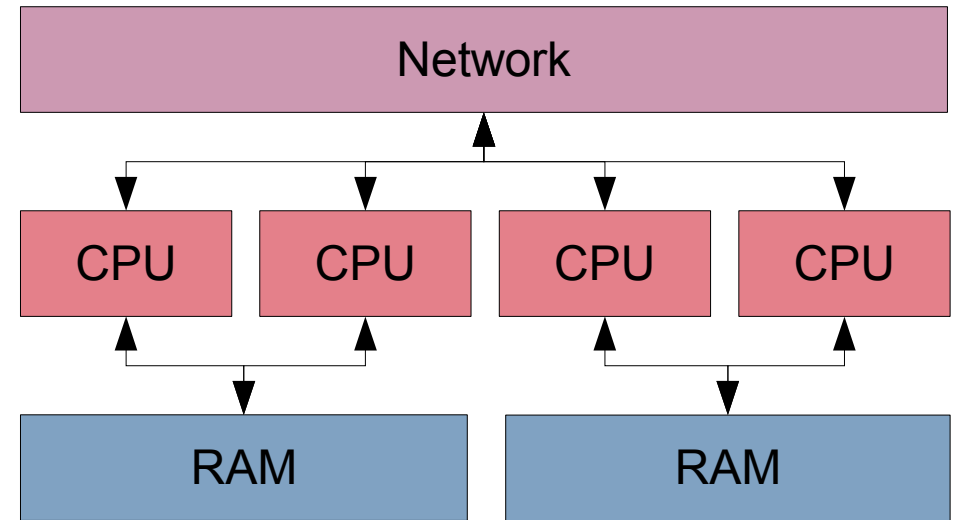
Also known as DM
= Distributed Memory.

Memory Models: NUMA vs. Hybrid



NUMA = Non-Uniform Memory
Access = distributed memory

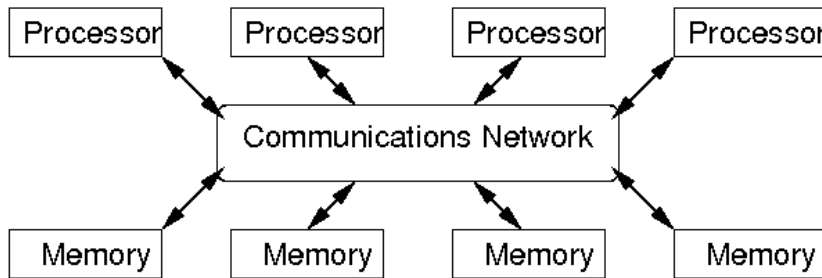
Some recent literature jumps straight to the hybrid architecture as the best example of a NUMA (thus losing the distinction between the two).



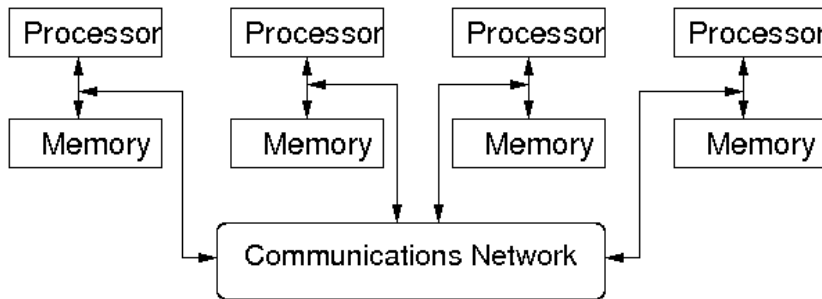
Hybrid (N)UMA (still NUMA, but...)

If there is a trend visible in the TOP 500, it's toward hybrid memory architectures (see <http://top500.org/>).

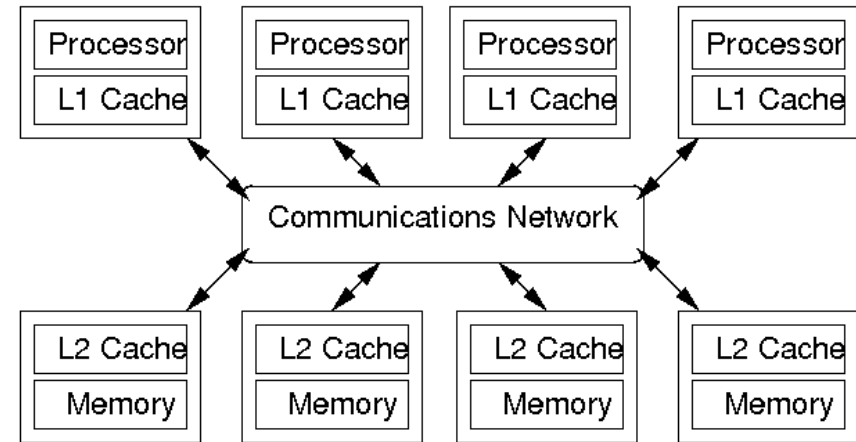
Memory Models: (N)UMA vs. cc(N)UMA



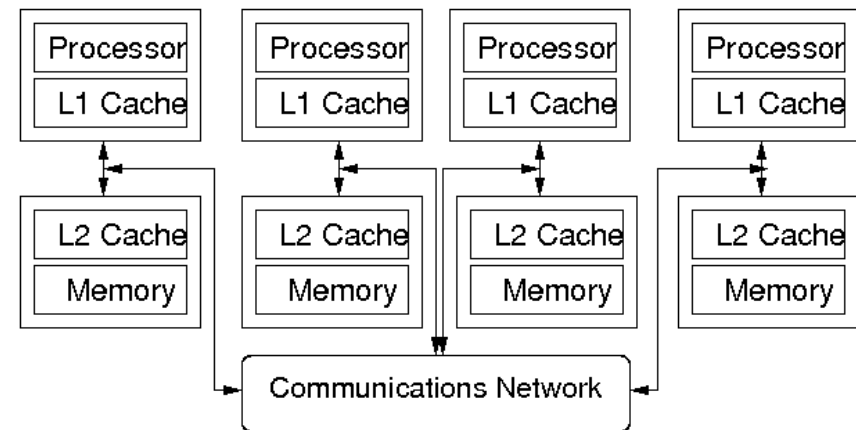
(a) UMA – Uniform Memory Access



(b) NUMA – Non-Uniform Memory Access



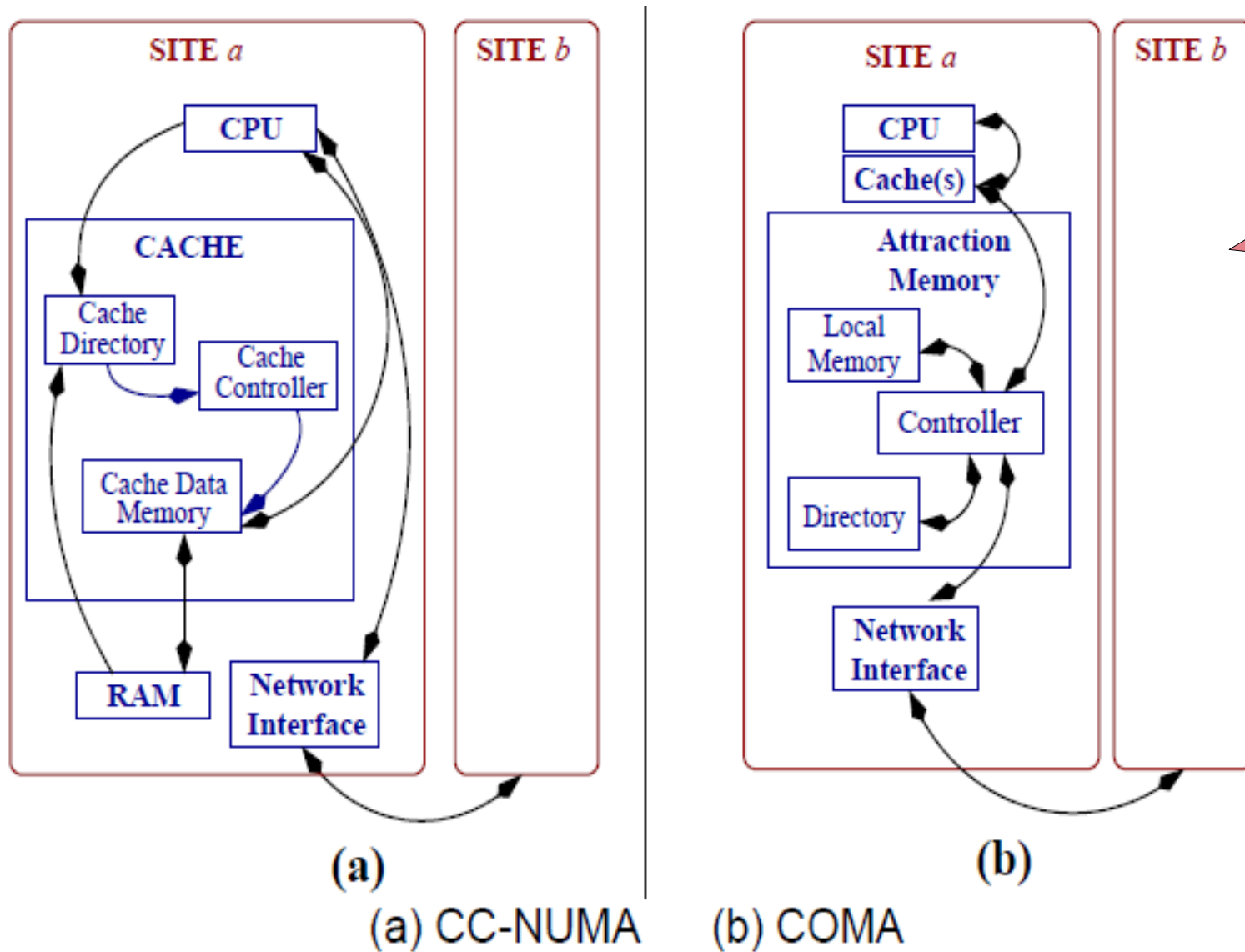
(c) CC-UMA – Cache Coherent Uniform Memory Access



(d) CC-NUMA – Cache Coherent Non-Uniform Memory Access

Sources: Ronald Moore, *SDAARC: A Self Distributing Associative Architecture*, Shaker Verlag, 2002

Memory Models: COMA



COMA = Cache
Only Memory
Architecture

Used
essentially
nowhere, but
a good idea
nonetheless...

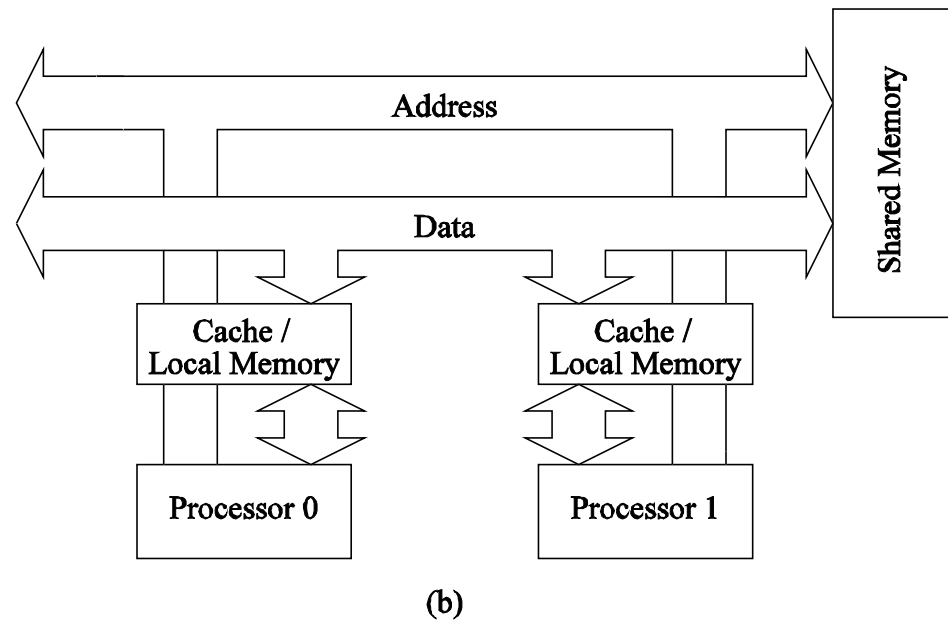
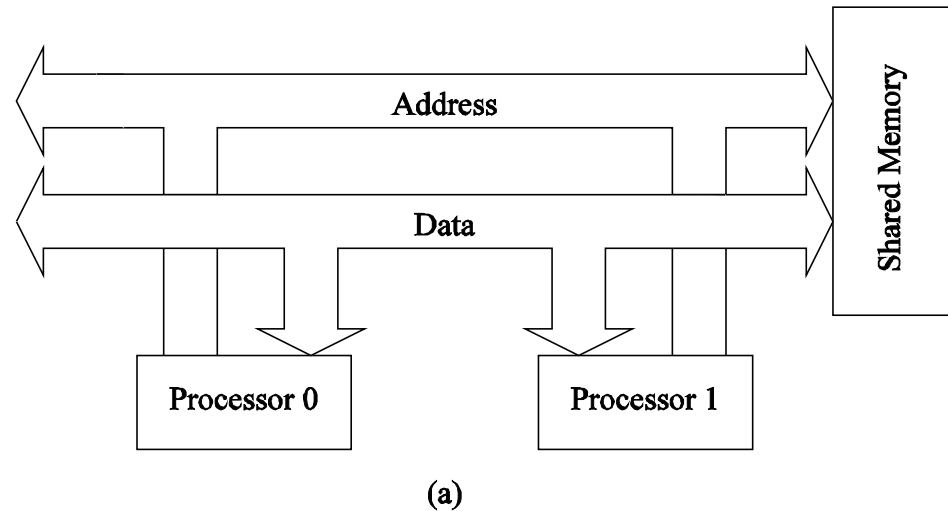
Sources: Ronald Moore, *SDAARC: A Self Distributing Associative Architecture*, Shaker Verlag, 2002

Under the Hood: Memory Networks (1)

Connecting CPUs & Memory

→ Bus

- ❖ Limited bandwidth
- ❖ Good cost scaling (cheap)
- ❖ Limited performance scaling
- ❖ Helps Cache Coherency (see below).



Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

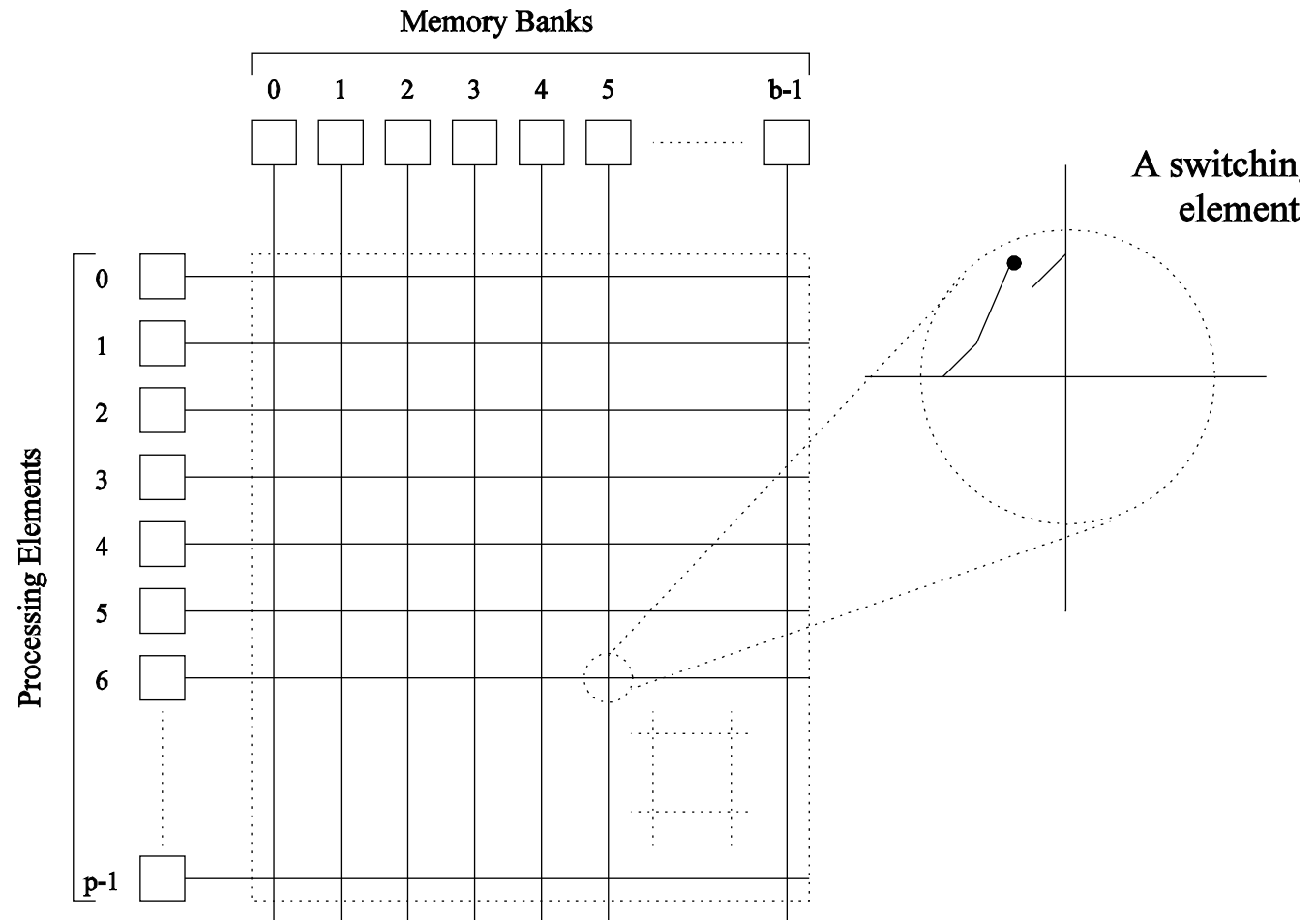
Under the Hood: Memory Networks (2)

Connecting CPUs & Memory

→ **Bus**

→ **Cross-bar**

- Good Bandwidth (non-blocking!)
- Good performance scaling
- Bad cost scaling



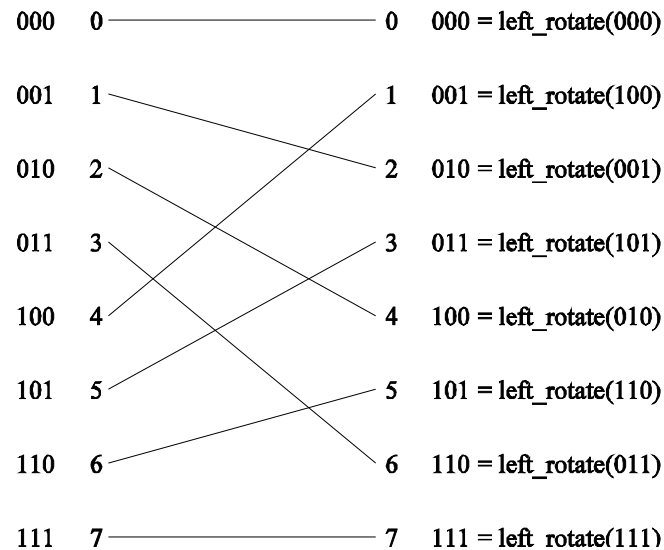
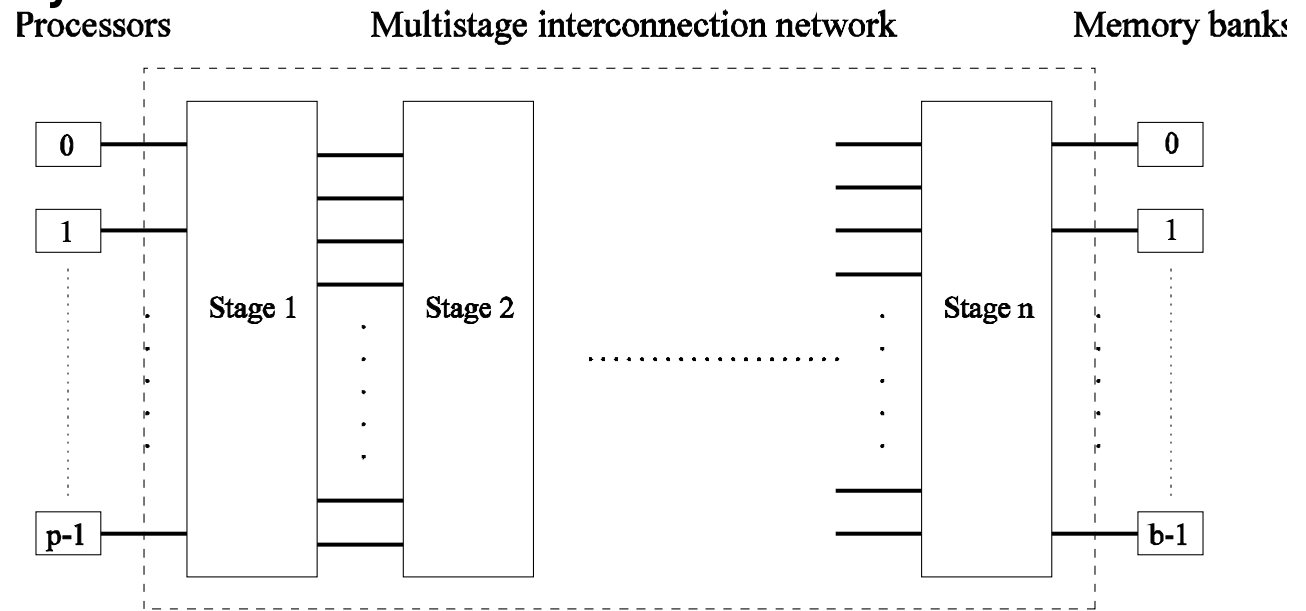
Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Under the Hood: Memory Networks (3)

Connecting CPUs & Memory

- **Bus**
- **Cross-bar**
- **Multistage Networks**
 - Good compromise between cost & performance

- Example:
Omega Network



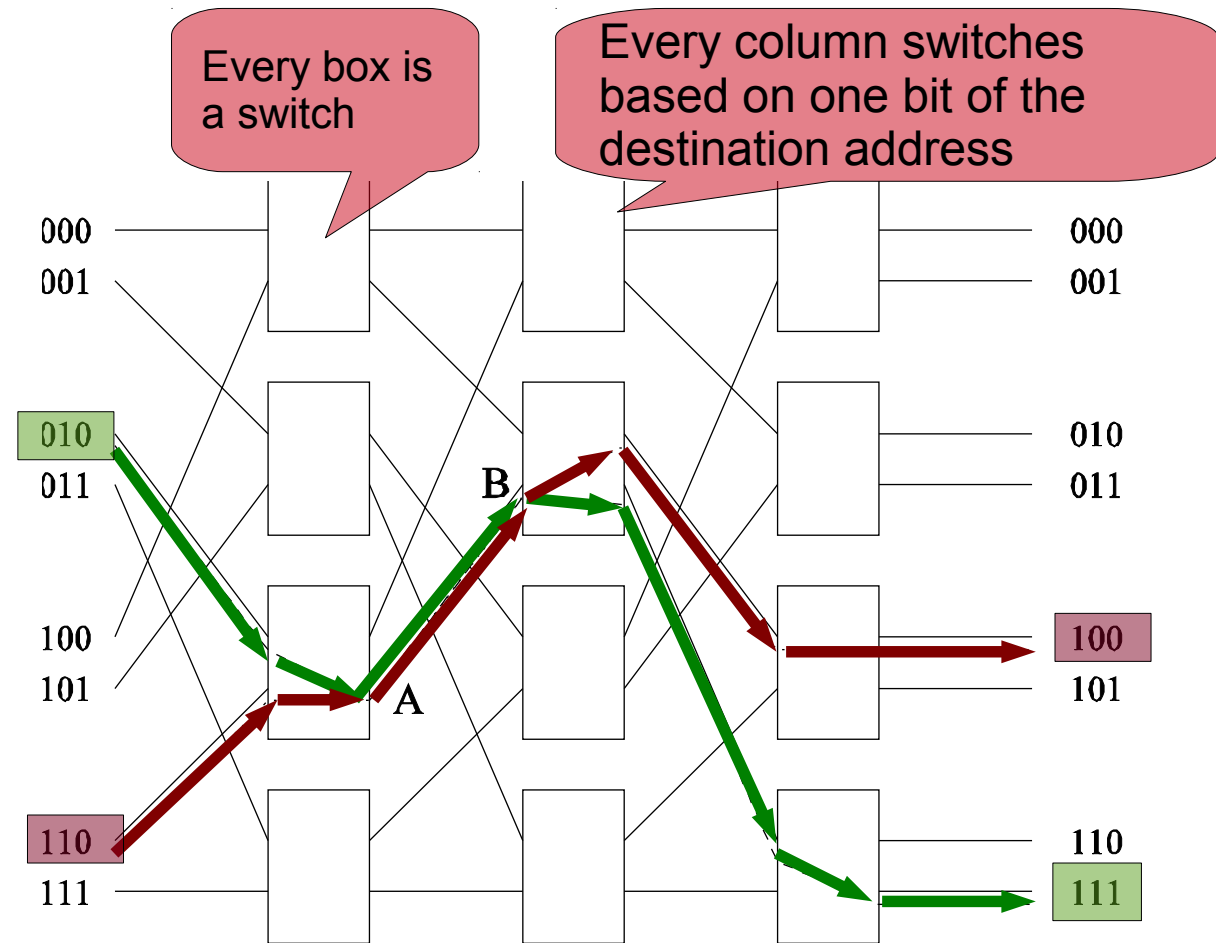
Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Under the Hood: Memory Networks (4)

Connecting CPUs & Memory

- **Bus**
- **Cross-bar**
- **Multistage Networks**
 - ♦ Good compromise between cost & performance
 - ♦ $p/2 \times \log p$ switching nodes...
 - ♦ Thus, cost grows as $(p \log p)$.
 - ♦ *Not* a non-blocking network!

The Ω -Network (a.k.a. *Perfect shuffle pattern*) shows up again and again... keep an eye out for it!



Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Under the Hood: Cache Coherence (1)

Cache Basics

- ➔ Caches work because of:
 - ◆ **Temporal Locality** – I'll probably keep using this data for a while.
 - ◆ **Spacial Locality** – I'll probably use it's data in it's neighborhood too.
- ➔ **Problem:** (False) Sharing of data leads to multiple (potentially) inconsistent copies of the data in the caches.
- ➔ **Solution:** The caches need to be aware of all (relevant) transactions → coherence.
- ➔ **Protocol determines actions.**
Alternatives:
 - ◆ **Write-through or Write-Back**
(Does data stay in the cache on a write – until the cache line is evicted, or is the RAM updated at once?)
 - ◆ **Write-Invalidate or Write-Update**
(What happens to the *other* copies when one CPU writes to RAM?)
 - ◆ **Snoopy** (for Buses/UMA) or **Directory-based** (for NUMA).

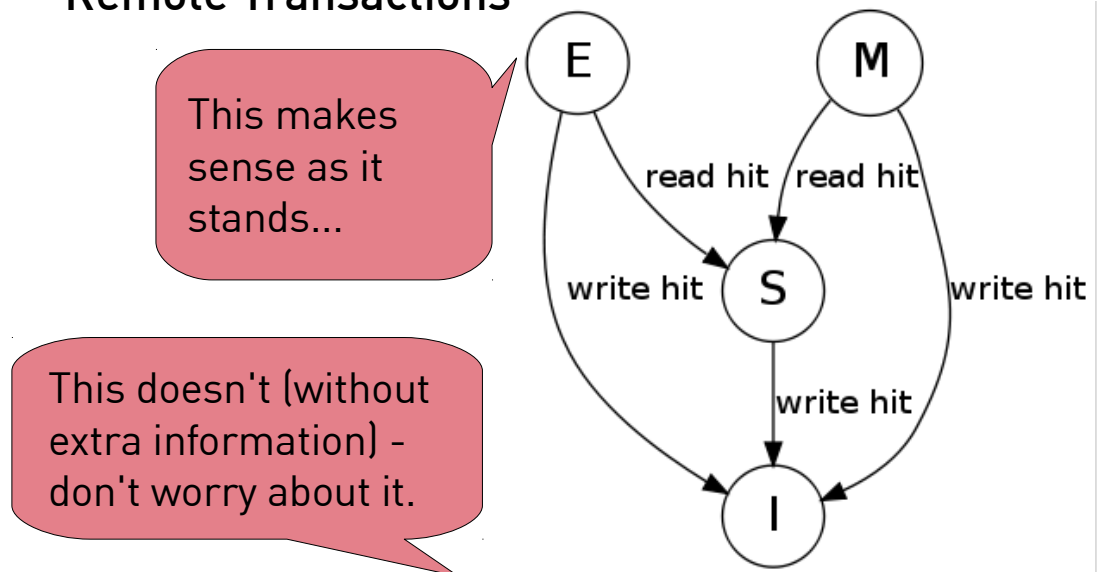
Under the Hood: Cache Coherence (2)

An Example: MESI

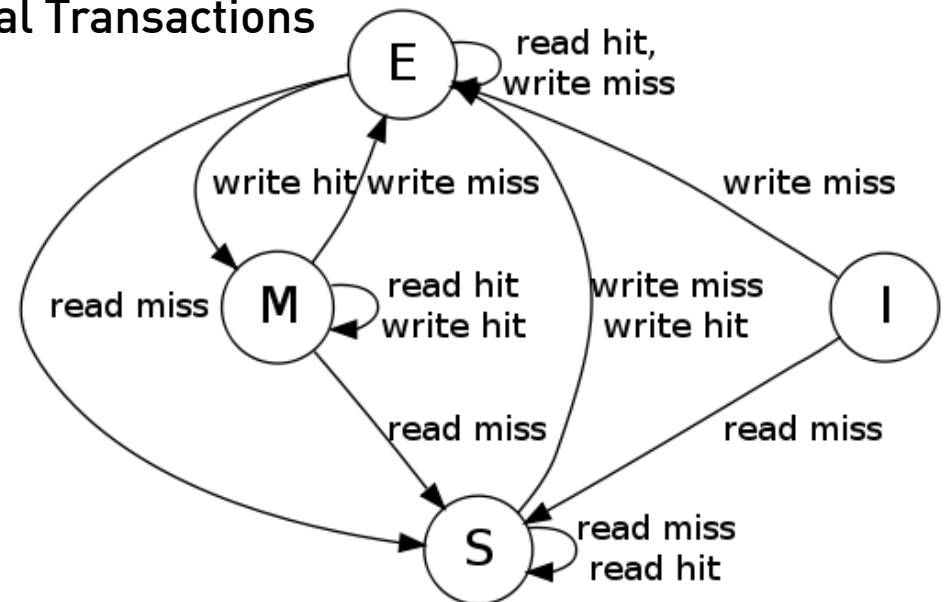
- ➔ Write-Back, Write-Invalidate, Snoopy Protocol
- ➔ Used e.g. in Intel Pentium.
- ➔ Each Cache Line is (at any given time) in one of 4 States:

- ♦ **Modified:** Has been written (will need to be written back).
- ♦ **Exclusive:** Only copy (not modified).
- ♦ **Shared:** *Potentially* one of many copies (not modified)
- ♦ **Invalid**

Remote Transactions



Local Transactions



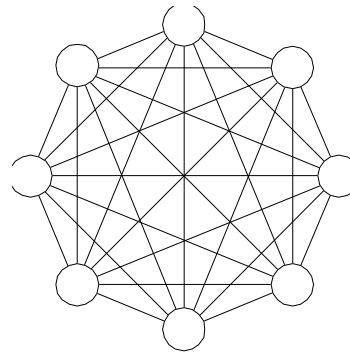
Outline of Chapter II – Models

- I Introduction
- II Models of Parallel Computing
- III Parallel Computation Design
- IV The Message-Passing Paradigm
- V The Shared Memory Paradigm
- VI Frontiers
- VII Summary

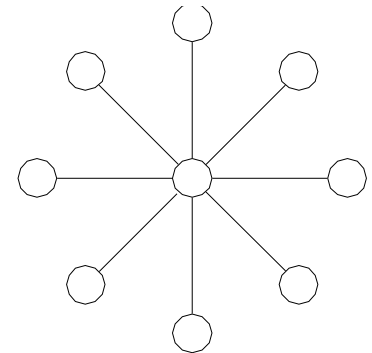
- A) Jargon & Terminology
- B) Taxonomies:
 - 1) Flynn's
 - 2) R. C. Moore's
- C) Models:
 - 1) Theoretical
 - 2) Hardware
 - a) Command Flow
 - b) Memory Models
 - 3) Communication
- D) Summary

Network Topologies (1)

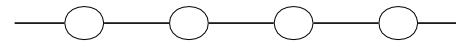
- Completely-connected
- Star-connected
- Linear-Array (with or without wrap-around)
- 2-D or 3-D Array (with or without wrap-around)



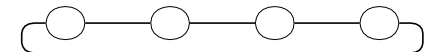
(a)



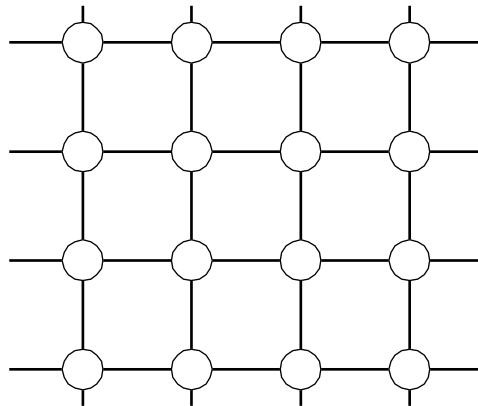
(b)



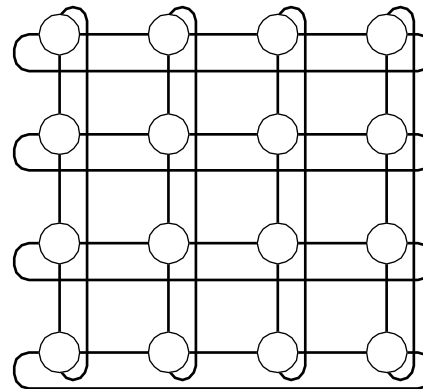
(a)



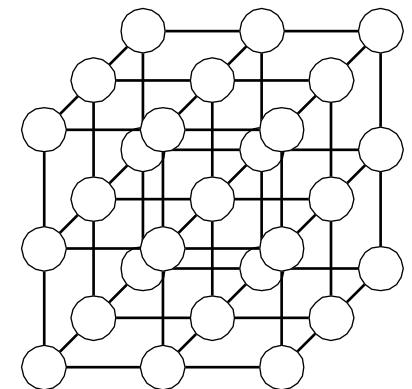
(b)



(a)



(b)

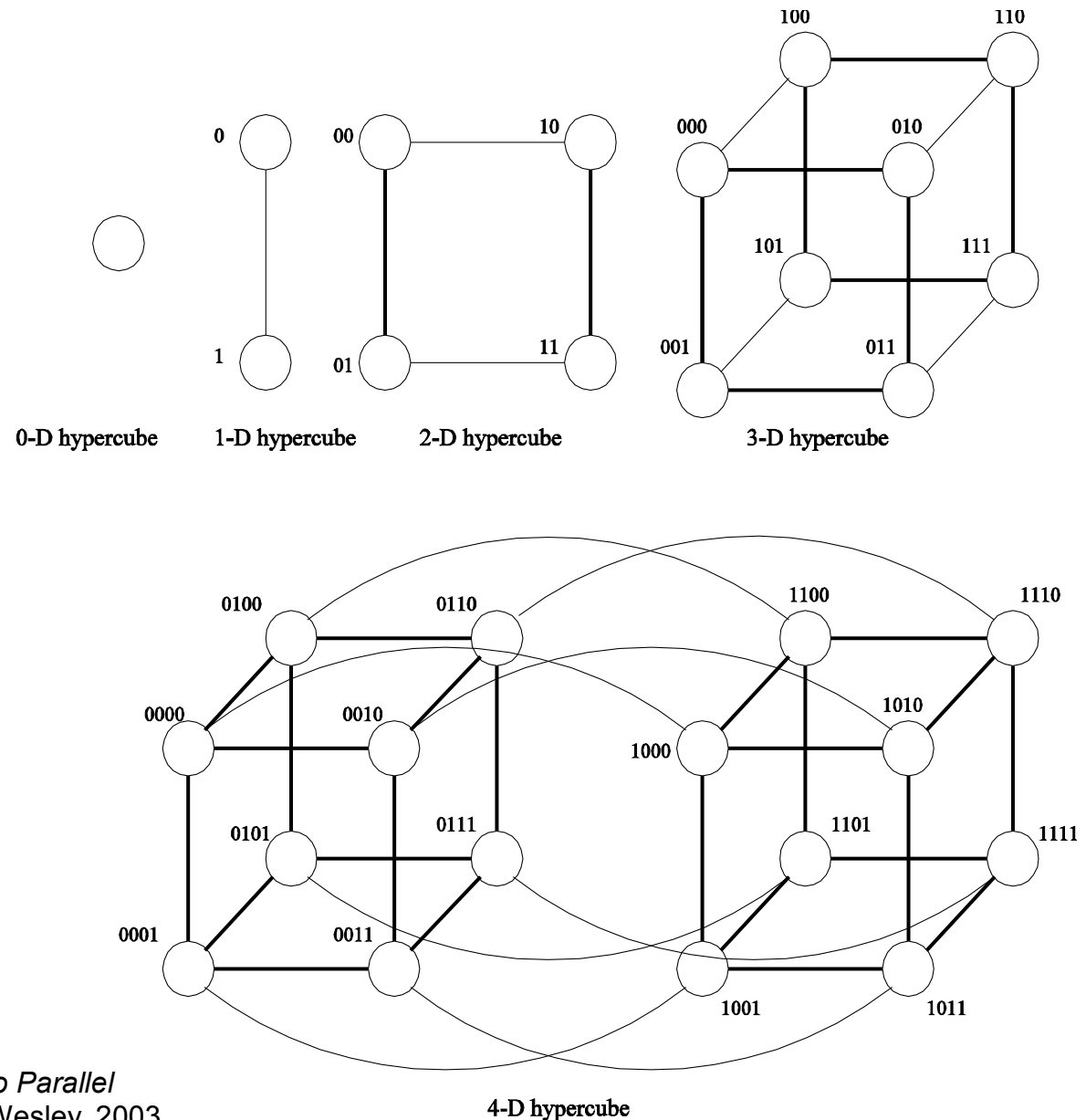


(c)

Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Network Topologies (2)

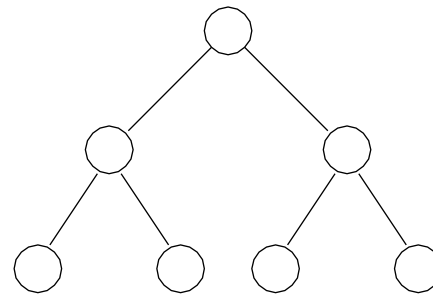
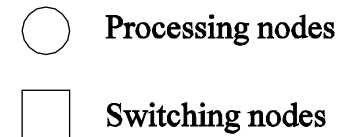
- Completely-connected
- Star-connected
- Linear-Array (with or without wrap-around)
- 2-D or 3-D Array (with or without wrap-around)
- Hypercube



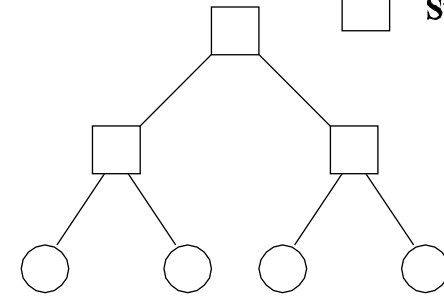
Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Network Topologies (3)

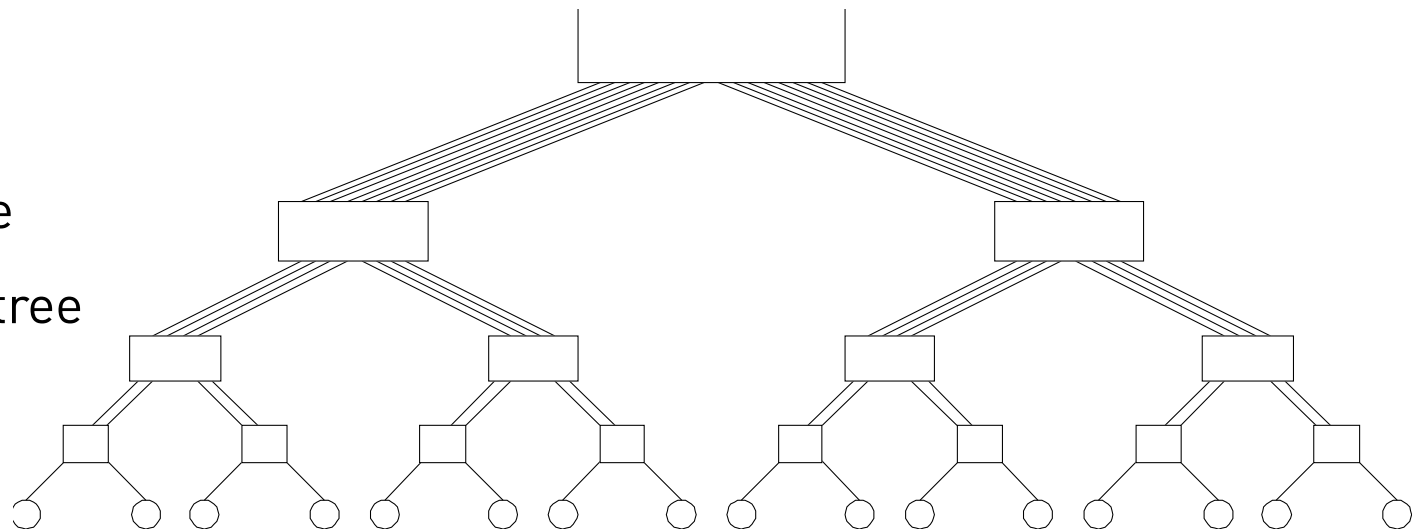
- Completely-connected
- Star-connected
- Linear-Array (with or without wrap-around)
- 2-D or 3-D Array (with or without wrap-around)
- Hypercube
- Trees:



(a)



(b)



- ➔ Static binary tree
- ➔ Dynamic binary tree
- ➔ Fat Tree

Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Network Topologies (4)

Evaluation Criteria

- **Diameter** = max distance between 2 nodes
- **Connectivity** = number of paths between 2 nodes.
- **Arc Connectivity** = minimum number of edges which, when removed, cut the network in two disconnected networks.
- **Bisection Width** = minimum number of edges which, when removed, cut the network in two equal halves.
- **Bisection-Bandwidth** = minimum bandwidth between two halves of the network.
- **Cost** = z.B. Number of connections...

Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

Network Topologies (5)

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p - 1$	$p(p - 1)/2$
Star	2	1	1	$p - 1$
Complete binary tree	$2 \log((p + 1)/2)$	1	1	$p - 1$
Linear array	$p - 1$	1	1	$p - 1$
2-D mesh, no wraparound	$2(\sqrt{p} - 1)$	\sqrt{p}	2	$2(p - \sqrt{p})$
2-D wraparound mesh	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p \log p)/2$
Wraparound k -ary d -cube	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	dp

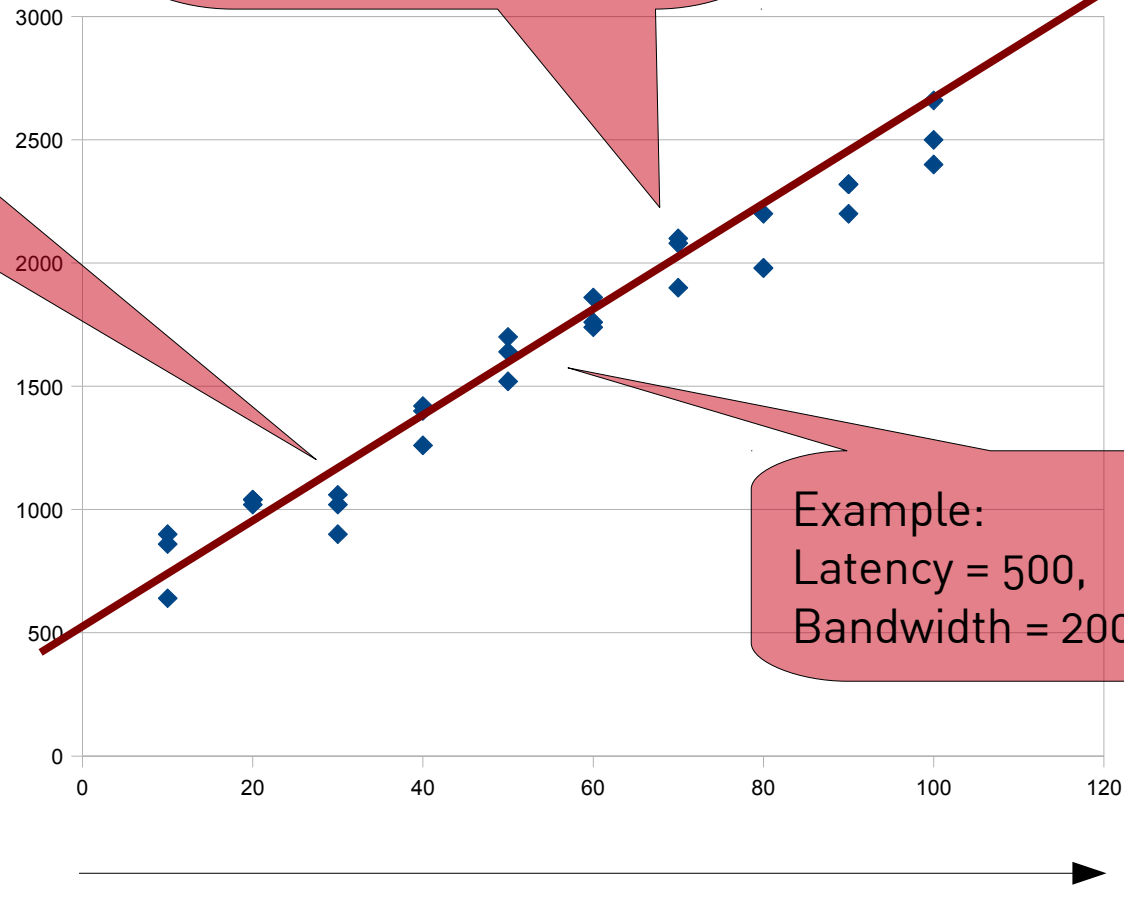
Source: Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, 2nd Edition, Chapter 2, Pearson, Addison Wesley, 2003.

A Temporal Model of Communication (1)

$$T_{\text{communication}} = \text{Latency} + \frac{\text{Message-Size}}{\text{Bandwidth}}$$

Implication:
1 message of size $2x$
is faster than 2
messages of size x
(iff Latency > 0 ...
and all data is ready to
send *now*).

Suppose communication
time was empirically
measured as shown...



Example:
Latency = 500,
Bandwidth = 200/sec

A Temporal Model of Communication (2)

- **Latency** is a fundamental problem whenever communication is necessary.
- **Solution 0:** No message-passing, only shared memory? Why is this not a solution?
- **Solution 1:** Low-latency networks? Why is this not a solution?
- **Solution 2: Latency Hiding:**
 - ➔ **Non-blocking send and receives:**
Sender-side and receiver-side buffers allow processors to get on with their work.
 - ➔ **Multithreading:** If one thread is blocked (waiting for a read or a write to finish), other threads can still execute other tasks.

Outline of Chapter II – Models

- I Introduction
- II Models of Parallel Computing
- III Parallel Computation Design
- IV The Message-Passing Paradigm
- V The Shared Memory Paradigm
- VI Frontiers
- VII Summary

- A) Jargon & Terminology
- B) Taxonomies:
 - 1) Flynn's
 - 2) R. C. Moore's
- C) Models:
 - 1) Theoretical
 - 2) Hardware
 - a) Command Flow
 - b) Memory Models
 - 3) Communication
- D) Summary

Summary (of Chap. 2)

A) Jargon & Terminology

B) Taxonomies

- Flynn: SISD, SIMD (!), MIMD (!), MISD (?)

Obligatory

C) Models

- *Theoretical:*

- PRAM
- Dataflow

Important for **finding** parallelism!

- *Control Flow:*

- Pipelines,
- Vector-Machines,
- VLIW...

Role models for *Task-Parallelism* &/or *Data-Parallelism*

- *Memory:*

- UMA, NUMA,
- CC-NUMA,
- Hybrid...

The basis for the shared memory paradigm

- *Communication Networks*

- ...and their evaluation criteria (not just bandwidth!)
- ...and the essential concept of *latency*.

The basis for message passing paradigm